A scalable and distributed dendritic cell algorithm for big data classification

Chelly Dagdia, Zaineb

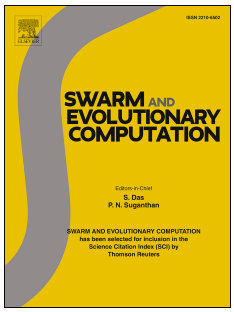
Published in:
Swarm and Evolutionary Computation

DOI:
10.1016/j.swevo.2018.08.009

Publication date:
2019

Citation for published version (APA):
Chelly Dagdia, Z. (2019). A scalable and distributed dendritic cell algorithm for big data classification. *Swarm and Evolutionary Computation*, *50*, [100432]. https://doi.org/10.1016/j.swevo.2018.08.009

Accepted Manuscript



A scalable and distributed dendritic cell algorithm for big data classification

Zaineb Chelly Dagdia

PII: S2210-6502(18)30200-1

DOI: 10.1016/j.swevo.2018.08.009

Reference: SWEVO 432

To appear in: *Swarm and Evolutionary Computation BASE DATA*

Received Date: 15 March 2018

Revised Date: 28 August 2018

Accepted Date: 31 August 2018

Please cite this article as: Z.C. Dagdia, A scalable and distributed dendritic cell algorithm for big data classification, *Swarm and Evolutionary Computation BASE DATA* (2018), doi: 10.1016/j.swevo.2018.08.009.

A Scalable and Distributed Dendritic Cell Algorithm for Big Data Classification

Zaineb Chelly Dagdia

Department of Computer Science, Aberystwyth University, Aberystwyth, United Kingdom
LARODEC, Institut Supérieur de Gestion de Tunis, Tunis, Tunisia,

Abstract

In the era of big data, scaling evolution up to large-scale data sets is a very interesting and challenging task. The application of standard biological systems in such data sets is not straightforward. Therefore, a new class of scalable biological systems that embraces the huge storage and processing capacity of distributed platforms is required. In this work, we focus on the Dendritic Cell Algorithm (DCA), a bio-inspired classifier, and its limitation when coping with very large data sets. To overcome this limitation, we propose a novel distributed DCA version for data classification based on the MapReduce framework to distribute the functioning of this algorithm through a cluster of computing elements. Our experimental results show that our proposed distributed solution is suitable to enhance the performance of the DCA enabling the algorithm to be applied over big data classification problems.

Keywords: Dendritic Cell Algorithm, Big Data, Distributed Processing.

1. Introduction

Under the explosive increase of global data, the term of big data is increasingly being used to refer to the challenges and benefits derived from gathering and processing enormous data [1]. Formally, big data is defined as the amount of data that exceeds the capabilities of a given system to process the data in terms

Email address: chelly.zaineb@gmail.com (Zaineb Chelly Dagdia)

of time and/or memory consumption [2]. Nowadays, big data is attracting much attention in a wide variety of fields such as social media [3], healthcare and government [4], financial businesses, or network applications. This is because of the progressive acquisition of a huge amount of data which becomes easily accessible and due to the availability of distributed platforms [5]. With these facilities, new opportunities for discovering new values from massive data sets can be sought, helping to gain an in-depth understanding of the hidden values, and also to incur new challenges. These challenges arise from data collection and data curation, i.e. content creation, selection, classification, transformation, validation, and preservation, to data analysis and data visualization [6]. These tasks become very difficult to achieve for most of the classical and state-of-the-art data mining and machine learning techniques and among these are methods which are based on bio-inspired approaches [7]. Focusing on machine learning and specifically bio-inspired techniques dedicated for such task, in literature, most of the used off-the-shelf bio-inspired algorithms and processing technologies cannot work efficiently and satisfactorily in the context of big data. Therefore, it becomes necessary to develop and create new techniques and technologies to enable enhanced decision making, insight discovery and process optimization.

In this sense, a new generation of robust fault-tolerant systems, based on parallel computing, has been established where the MapReduce framework [8, 9] is the most representative one. In literature, several research papers have focused on the parallelization of data mining and machine learning techniques based on the MapReduce model. These techniques such as clustering techniques [10], classification techniques [11], mining techniques [12] and dimensionality reduction techniques [13] have proved that the distribution of the data and the processing under a parallel computing infrastructure is very useful for speeding up the knowledge extraction process. Among the machine learning techniques which are based on bio-inspired MapReduce approaches, we mention those mechanisms inspired by biological evolution and dedicated for either data preprocessing or to deal with imbalanced data sets in classification problems. For data preprocessing, classical evolutionary approaches have been successfully used to preprocess

data [14] but only data with moderated size. The reason behind this is the excessive increment of the individual or the chromosome size which disables the algorithms to provide a preprocessed data set in a reasonable time when dealing with very large problems. Therefore, distributed evolutionary big data models have been proposed to deal with the feature space, i.e. feature selection [15] and feature weighting [16], and with instance reduction, i.e. instance selection [17] and instance generation [18]. On the other hand, regarding the problem of imbalanced data sets, this scenario appears frequently in the classification problem field. This issue appears mainly when examples of one class significantly outnumber the examples of the other one [19]. To deal with this problem, evolutionary algorithms for imbalanced big data sets have been proposed [20, 21, 22]. Technically, these MapReduce evolutionary algorithms are based on resampling techniques [23] to study the effect of changing the class distribution. Specifically, these techniques are based on undersampling methods to create a subset of the original data set by eliminating instances which are usually presenting the majority class. Other bio-inspired distributed approaches, particularly evolutionary approches, were proposed in literature such as [24] where the genetic algorithm was modeled into the MapReduce paradigm while demonstrating the convergence and scalability of the proposed parallel algorithm. Some other distributed methods inspired by swarm intelligence were proposed such as [25] where a parallel ant colony optimization algorithm was proposed to deal with the large number of ants and iterations required by the algorithm as these consume more time and resources, and the work proposed in [26] that presents a MapReduce particle swarm optimization algorithm (PSO) capable of handling the greedy resources expressed by the individual function evaluations operations used by the classical PSO algorithm. Another category of bio-inspired methods is artificial immune systems. Within this category, lately, some distributed works have been proposed in literature; among these we mention [27] a work proposing a decentralized and fault tolerant immune framework for the distribution of security information for industrial networks. It is worth mentioning that in literature, there are more variants of parallel evolutionary

algorithms than the bio-inspired swarm intelligence techniques or the ones based on artificial immune systems.

Parallel evolutionary algorithms have been more developed, studied, and applied in literature, and a review of these and their corresponding history can be found in [28]. The survey in [29, 30] contains detailed overviews of parallel evolutionary algorithms and their characteristics. A broader scope on parallel evolutionary algorithms can be found in [31, 32] where several parallel variants of evolutionary algorithms are covered such as genetic programming, evolution strategies, ant colony optimization, swarm intelligence algorithms, estimation-of-distribution algorithms, scatter search, and simulated annealing.

In this paper, we aim to further enable algorithms inspired by nature to be applied on big data; specifically algorithms inspired by the immune systems. In this concern, we focus on the Dendritic Cell Algorithm (DCA) [33] which is a bio-inspired algorithm that has caught the attention of many researchers due to its worthy characteristics as it exhibits numerous interesting and advantageous features for classification problems [34]. Despite the emergence of the DCA, in the current literature, the practical application of the algorithm was limited to problems with moderated size only. The reason behind this arise from the necessity to use an antigen multiplier to generate at once several copies of antigens, referring to data instances, to process them in turn to finally perform the classification task. More precisely, the DCA requires multiple instances of identical antigens, so processing across a population can be performed in order to generate the classification results for each antigen. The antigen multiplier is implemented to overcome the problem of "antigen deficiency", that is, insufficient antigens are supplied to the DC population. As one antigen can be generated from each data instance within a data set, the antigen multiplier can make several copies of each individual antigen which can be fed to multiple DCs. However, as the number of data instances is increasing this task becomes challenging and this is where the DCA inadequacy arises. It is quite unmanageable to generate the set of all antigen copies based on the huge number of data instances due to hardware and memory constraints. This leads us to progress in

the field being disjointed and, ultimately, improve the performance of the DCA to be successfully applied to big data applications.

In this work, we propose a novel efficient distributed dendritic cell algorithm for large-scale data sets which solves the standard DCA mentioned computational inefficiencies and its restriction to be only applied to moderated size data sets. Our novel DCA version is characterized by its distributed implementation design based on both Scala and the MapReduce Apache Spark framework [35]. Developing a distributed schema based on MapReduce for the dendritic cell algorithm motivates the global purposes of this work which are (i) to enable the DCA to deal with big data classification problems (ii) to illustrate the scalability of the proposed schema (iii) to analyze the behavior of our proposed solution within a distributed environment particularly in terms of classification performance and (iv) to investigate the insights tied to the parallelization of the algorithm.

The rest of this paper is organized as follows: Section II provides some background material about the dendritic cell algorithm and the distributed processing framework including the MapReduce paradigm. Section III introduces our novel distributed DCA for large-scale data classification. The experimental setup is introduced in Section IV. The results of the performance analysis are given in Section V and the conclusion is given in Section VI.

2. Background

In this Section, we provide background information about the dendritic cell algorithm. We, also, discuss a set of well-known distributed processing frameworks including the MapReduce paradigm.

2.1. The Dendritic cell algorithm

The Dendritic Cell Agorithm (DCA) [36] is based on an abstract model of biological Dendritic Cells (DCs). It is a population-based algorithm where each agent is represented as a cell (DC). Each cell has the capacity to collect data

instances termed antigens representing the data to be classified. Formally, the DCA has two main inputs which are the input data in the form of a set of signals and antigens. DCA has to classify each antigen either as normal or as anomalous. To do so, the algorithm goes through four main phases.

Through the *initialization phase*, the first step, DCA performs data preprocessing where feature selection and signal categorization are achieved. More precisely, DCA selects the most important features (attributes) from the input data set and assigns each selected attribute to its specific signal category; either as a Safe Signal (SS), as a Danger Signal (DS), or as a Pathogenic Associated Molecular Patterns (PAMP) signal. To achieve this task, some DCA works deal with involving the user to select or extract the most interesting features and map them into their appropriate signal categories where some other works call for some statistical approaches and dimensionality reduction techniques such as the principal component analysis [37].

Through the second DCA step, which is the *detection phase*, the algorithm prepares a signal database where its rows represent the antigens to be classified and the attributes represent the signals, i.e. SS, PAMP and DS. The attribute values, for each antigen, are calculated based on specific mathematical formulas [37]. Meanwhile, the DCA prepares a population of artificial DCs and the set of antigens copies using an antigen multiplier. The antigen multiplier which is a discrete number m is used to copy each data instance (antigen) m times. Using the prepared signal database, the algorithm processes its input signals to get three cumulative output signal values known as the costimulatory molecule signal value "CSM", the semi-mature signal value "$smDC$" and the mature signal value "mDC". These three output signals perform two roles which are, first, to determine if an antigen type is anomalous and, second, to limit the time spent sampling data. Each DC in the population is assigned a migration threshold value "mt". If the value of CSM exceeds mt then the DC stops sampling antigens and signals; else the algorithm continues sampling and, also, keeps calculating and updating the values of CSM, $smDC$ and mDC [37].

The *context assessment phase*, the third step, is where the DCA forms a cell

context that is used to perform antigen classification. In fact, the cumulative output signals of both $smDC$ and mDC are assessed and the one that has a higher output signal is the one that becomes the cell context; either 1 or 0. The derived value for the cell context is used to derive the nature of the response by measuring the number of DCs that are fully mature. This generated number is represented by the Mature Context Antigen Value "$MCAV$". The $MCAV$ is calculated by dividing the number of times an antigen appears in the mature context by the total number of presentation of that antigen, i.e. the antigen multiplier.

Finally, to perform its binary *classification task*, the DCA compares the $MCAV$ of each antigen to an anomalous threshold. Those antigens whose $MCAVs$ are greater than the anomalous threshold are classified into the anomalous category while the others are classified into the normal one. For the DCA pseudo-code, we kindly invite the reader to refer to [37].

In literature, several studies have been conducted to study the DCA algorithmic details and to address and resolve its shortcomings by proposing new modified DCAs. The main DCA versions tend to either modify the algorithm principals by simplifying it either by removing or replacing some of its components, or they tend to hybridize the DCA with other mathematical theories; mainly to deal with the encountered imprecision in the DCA concepts. Therefore, the DCA has undergone many revisions since its original inception, resulting in multiple versions of the algorithm. Some DCA versions improve the DCA algorithmic steps by using a deterministic process for instance, where some other versions investigate and improve the algorithm preprocessing phase by using mathematical models such as rough set theory and fuzzy rough set theory, while some other versions improve the algorithm behavior as a classifier by using database maintenance techniques, fuzzy set theory and fuzzy clustering techniques. A full review of all these methods can be found in [37]. Recently, in [38] a revision of the DCA was made via a new approach inspired by purely functional programming; aiming to introduce the DCA to a new audience within computer science. Despite the development of several DCA versions, the appli-

cation of the algorithms remained limited as being not adapted to deal with the big data context.

2.2. Distributed processing frameworks

In the context of big data, it becomes mandatory to develop and create new techniques and technologies to enable enhanced decision making, insight discovery and process optimization. In this sense, a set of technologies [39] has emerged to deal with big data where common solutions are those based on parallel computing such as the Message Passing Interface (MPI) model [40]. Challenges at this point are mainly tied to the data access and to the simplicity in developing software with respect to the requirements and restrictions of the available programming schemes [41]. For instance, classical algorithms need all the data to be loaded into the main memory. This presents a technical barrier in big data as the input data are often stored in various locations inferring an intensive network communication and other input/output costs; and even if we can afford this then it is essential to offer an extremely large main memory to hold all the preloaded input data for the computation.

To deal with these issues, a new generation of robust fault-tolerant distributed systems has been established. These processing frameworks can be grouped by the state of the data they can handle. More precisely, some of these frameworks can process data in a batch-only schema where the processing distributed system operates over a large and static database, and then at a later stage returns the result(s) when all computations are finished. Hadoop[1] was the first big data processing framework that is dedicated for batch processing. Hadoop offers a scalable and a highly reliable distributed processing of large data sets via the use of simple programming models. With the ability to be built on clusters of commodity computers, Hadoop provides a cost-effective solution for storing and processing structured, semi- and unstructured data with no format requirements. The core of Hadoop is the MapReduce model. It is a program-

[1] http://hadoop.apache.org/

ming paradigm that allows for massive scalability across a very large number of servers in a Hadoop cluster. Further discussions about MapReduce will be give later in this section. Other systems handle data in a stream-only way where computations are processed over data as they enter the system, i.e. calculations are applied to each individual data item as it enters the framework. Among the most popular stream workload systems are Apache Storm[2] and Apache Samza[3]. Some other frameworks are hybrid systems as they can process data in both of the batch and the stream ways. These hybrid frameworks simplify diverse processing requirements by allowing the same or related components and APIs to be used for both types of data. Apache Spark[4] and Apache Flink[5] are considered to be the most popular streaming processing frameworks being used today. In this paper, we will mainly focus on Apache Spark which is a cluster computing framework originally developed in the UC Berkeley AMP Lab for large-scale data processing that improves the efficiency by the use of intensive memory. It is characterized by its performance, its highly transparency for programmers which allows to parallelize applications in an easy and comfortable way and its open source nature. The choice of this specific framework is mainly based on the following reasons (i) to make our proposed solution more general as it is based on a hybrid distributed system (ii) Spark offers incredible speed advantages, trading off high memory usage (iii) Spark is among the very well-known, mature, and well-tested frameworks in comparison to others which are more niche in their usage and which are still in their early days of adoption[6].

Apache Spark relies on a data structure known as the Resilient Distributed Data set (RDD). This is a read-only multiset of data items that is distributed over the entire cluster of machines. RDDs operate as the working set for distributed programs, offering a restricted form of distributed shared memory. Spark also uses the MapReduce concept [42, 8, 9] which was introduced by

[2] http://storm.apache.org/
[3] http://samza.apache.org/
[4] https://spark.apache.org/
[5] https://flink.apache.org/
[6] https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared

Google in 2004. MapReduce is a programming model that offers a simple but robust environment to process large data sets over a cluster of machines independently from the underlying hardware and/or software. In this model, the user has to specify the computation in terms of a map function and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks [8, 9]. The map phase allows different points of the distributed cluster to distribute their work. In this phase, the input data set is processed producing some intermediate results. On the other hand, the reduce phase is designed to reduce the final form of the clusters results into one output. A flowchart of the MapReduce framework is presented in Figure 1.

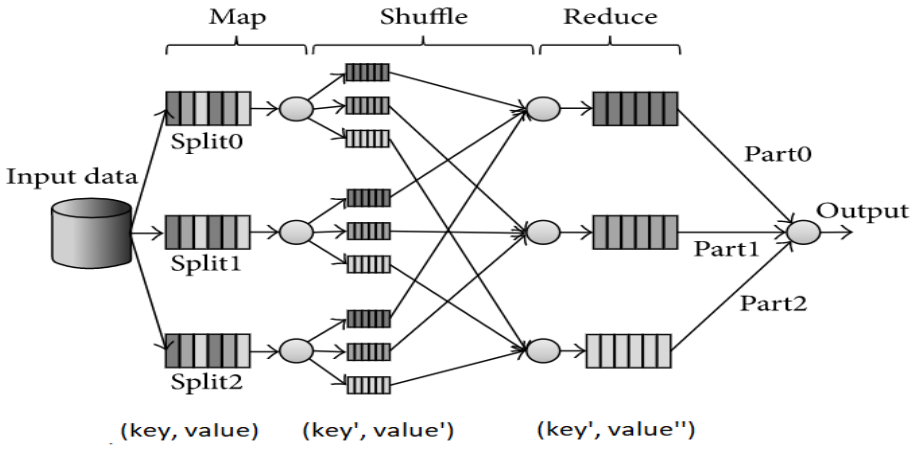


Figure 1: The process of the MapReduce framework.

Formally, the MapReduce model is based on a basic data structure known as the key-value ⟨k, v⟩ pair. Each of the MapReduce steps has its own ⟨k, v⟩ pairs as input and output. In the map phase, each application of the map function receives a single ⟨k, v⟩ pair as input and generates a set of intermediate ⟨k', v'⟩ pairs as output. This is represented as follows:

$$map\langle k, v\rangle \rightarrow \{\langle k', v'\rangle, \ldots\}. \quad (1)$$

Then, MapReduce merges all the values associated with the same intermediate key as a list. The reduce phase takes that list as input for producing the final values. This is represented in the following form:

$$reduce\langle k'; \{v', \ldots\}\rangle \rightarrow \langle k'; v''\rangle. \quad (2)$$

All of the map and the reduce operations run in parallel within a MapReduce program. First, all of the map functions are run in an independent way. Meanwhile, the reduce operations wait until their respective maps are completed. Then, they process different keys concurrently and independently. The inputs and outputs of a MapReduce job are stored in an associated Distributed File System (DFS) that is accessible from any computer of the used cluster.

As previously mentioned, there are several open source Big Data processing frameworks in the market and among these, we have mentioned the most popular ones. Yet, it is worth mentioning that the best fit for choosing a specific framework always depends upon the state of the data to process, how time-bound the user requirements are, and what kind of results the user is interested in. As previously highlighted, in this paper, our choice is focused on the use of Apache Spark.

3. The Distributed dendritic cell algorithm

In this Section, we present our newly proposed distributed dendritic cell algorithm. Our proposed solution, dubbed "Sp-DCA", is characterized by its distributed implementation design with respect to the Spark framework for a parallel and in-memory processing task. Firstly, we argue the motivation that justifies the development of our proposed Sp-DCA solution by pointing out the standard DCA computational inefficiencies restricting it to be only applied to databases with moderated size. Then, we detail the proposed Sp-DCA solution in depth as an efficient bio-inspired technique dedicated for big data classification.

3.1. Motivation and problem statement

As previously mentioned, the DCA application was limited to problems with moderated size only. This is explained by the fact that the algorithm applies an antigen multiplier (m) for the purpose of classification. Each data instance (x_i) is copied (m) times generating $[x_i] * m$ antigens. With the size of the data set (N), a total of $[x_i] * m * N$ antigens will be generated from the whole input data set. Yet, as the number of data instances is increasing this task becomes challenging; and this highlights the DCA main inadequacy to be applied to large data sets. It is quite infeasible to generate all antigen copies with respect to the huge number of data instances due to hardware and memory constraints.

To deal with this, the parallelization as well as the distribution of workload in various sub-jobs may ease the enumerated problems which are tied to hardware, runtime and memory consumption. To tackle this challenge, we have to create an efficient DCA design that takes advantage of parallelization schemes, i.e. the MapReduce model, as justified in Section 2.2. The designed framework should enable DCA to be applied with data sets with a very large number of antigens. Furthermore, the proposed solution should guarantee that the objectives of the DCA are maintained, so that, it should provide satisfactory classification accuracy.

3.2. Sp-DCA: the proposed approach

To deal with high dimensional data sets it appears mandatory to store all the data in a distributed environment and ensure computations in a parallel way. With respect to this, we first partition the entire DCA algorithmic processes into elementary tasks, each executed independently, and then conquer the intermediate results to finally acquire the ultimate output; the classes of the antigens.

3.2.1. General model formalization

For antigen classification, Sp-DCA has to go through its distributed phases run on the original high dimensional input database which corresponds to the

data stored in the given Distributed File System (DFS) as a single file. To operate on the given DFS in a parallel way, a Resilient Distributed Data set (RDD) is created. We may formalize the latter as a training set, of a determined size N, which corresponds to the antigen data set defined as T_{RDD}, where universe $U = \{x_1, \ldots, x_N\}$ is the set of antigen identifiers, the attribute set $C = \{c_1, \ldots, c_V\}$ contains every single feature of the T_{RDD} and the decision attribute D of our learning problem corresponds to the class label of each T_{RDD} sample. As Sp-DCA is based on the standard DCA concepts, and since DCA is applied to binary classification problems; then our developed Sp-DCA is, also, applied to two-class data sets. Therefore, the decision attribute, D, of the input database of our Sp-DCA has binary values d_k: either the antigen is collected under safe circumstances reflecting a normal behavior (classified as normal) or the antigen is collected under dangerous circumstances reflecting an anomalous behavior (classified as anomalous). The decision attribute feature D is defined as follows: $D = \{d_{normal}, d_{anomalous}\}$. The universe set U presents the pool from where the antigens will be multiplied by an antigen multiplier m generating a pool of antigens $AntigensPool = \{an_{1_i}, \ldots, an_{1_m}, \ldots, an_{N_i}, \ldots, an_{N_m}\}$.

In order to make our algorithm scalable with the high number of both training data and antigens and within the Apache Spark perspective, Sp-DCA partitions the given T_{RDD} into p data blocks based on splits from the universe set U. Indeed, Sp-DCA creates an RDD from the generated antigens pool, $AntigensPool_{RDD}$, and splits it into a a number of disjoint subsets. Both of these RDDs are accessible from any computer of the cluster independently of their size. In such a way, $T_{RDD} = \bigcup_{i=1}^{p} \bigcup_{j=1}^{N} (x_j) T_{RDD_{(i)}}$ and $AntigensPool_{RDD} = \bigcup_{i=1}^{a} \bigcup_{x,y=1}^{m,N} (an_{x,y}) AntigensPool_{RDD_{(i)}}$. To ensure scalability, rather than applying Sp-DCA to T_{RDD} including the whole antigens from the universe set U and to the $AntigensPool_{RDD}$ including all the copies of antigens, the distributed algorithm will be applied to every single $T_{RDD_{(i)}}$ and to every single $AntigensPool_{RDD_{(i)}}$ that at the end all the intermediate results will be gathered from the different p and a partitions. In such a way, we can guarantee that Sp-DCA can be applied to a computable number of antigens

while dealing with the large number of the antigens copies and hence solving the standard DCA computational inefficiencies.

3.2.2. Algorithmic details

Sp-DCA follows the same DCA standard algorithmic steps previously discussed in Section 2.1. Therefore, the algorithm goes first through an initialization step followed by a detection phase then a context assessment phase to finally perform its classification phase. A flowchart of the proposed solution is given in Figure 2. In what follows, we will detail each of these Sp-DCA distributed algorithmic steps.

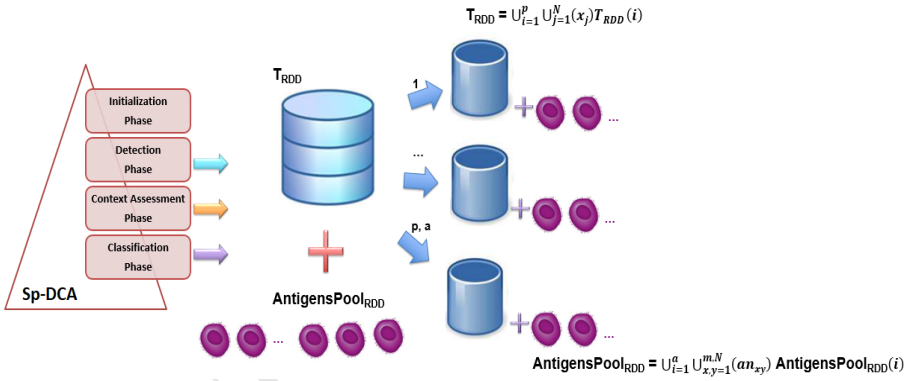


Figure 2: The Sp-DCA flowchart.

Initialization Phase. Just like the DCA, the application of Sp-DCA often requires a data pre-processing phase to appropriately map a given problem domain to the input space of the algorithm. The algorithm initialization phase enrolls two key tasks namely dimensionality reduction and signal categorization. Through this phase, the most important features are either selected or extracted from the input T_{RDD} and each is assigned to its specific signal category; either as SS, as DS or as a PAMP signal. Each attribute is mapped as a signal category based on its immunological definitions [37]. To perform the initialization phase, Sp-DCA involves the user or the expert to select the most interesting features and to map them into their appropriate signal categories.

A detailed analysis of this phase and the development of a distributed data pre-processing method (either a new parallel feature selection or a new parallel feature reduction technique) is out of scope for this paper.

Detection Phase. Throughout the detection phase, DCA has to generate a signal data set by combining the attribute signals defined in the first phase, initialization step, with the antigens. The induced signal database rows represent the antigens to classify and the attributes represent the three signals; PAMP, SS and DS. The signal attribute values for each antigen in the signal base are calculated as follows:

- **Process for calculating PAMPs and SSs:** As both PAMPs and SSs are positive indicators of an anomalous and normal signal, respectively, one attribute is used to form both of them. In this way, we contrive the scenario where the algorithm is given a context of either PAMP or SS. Using one attribute for these two signals requires a threshold level to be set which is considered to be the median value of the selected PAMP/SS attribute. For each attribute value in the $T_{RDD_{(i)}}$ partition, Sp-DCA has to determine if it is a PAMP or a safe signal. If the attribute value is greater than the median, then this value is used to form a safe signal. The absolute distance from the mean is calculated and attached to the safe signal value and the PAMP signal value takes 0 (and vice versa).

- **Process for calculating DSs:** As the danger signal is less than certain to be anomalous then a combination of several attributes is used resulting in a value that may be used as anomalous. To do so, the mean value for each DS attribute set within the used $T_{RDD_{(i)}}$ partition is required. To proceed with this, the absolute distance between the attribute values and the calculated means is generated. The calculated distance values are used in a further calculation to form the single value for the DS. This value is the mean value of the absolute distances calculated across the used number of features representing the DS. This process is applied for all entries of the selected attributes and for all the p $T_{RDD_{(i)}}$ partitions.

To achieve all these calculations in a distributed way and by working on the p generated partitions, first, Sp-DCA has to prepare the T_{RDD} input data in a specific format. More precisely, Sp-DCA will gather all the $[x_i]$ values belonging to a same $T_{RDD_{(i)}}$ column and create their corresponding column index. To do so, Sp-DCA processes the $zipWithIndex()$ operation that returns a new list containing pairs consisting of all elements of this list paired with their index, and the $swap()$ operation that swaps the first and second elements of a pair with each other, to generate the key reflecting the $column_{index}$. After that the $groupByKey()$ function which groups all the values related to a given key is applied to set the list of the $[x_i]$ objects having the same key, $column_{index}$, and hence the value is defined. These $\langle key, value \rangle$ pairs define the $ColumnsIndxVals$ output variable. The pseudo-code related to this distributed job is highlighted in Algorithm 1.

Algorithm 1 List the columns indexes and values

Input: T_{RDD}

Output: $ColumnsIndxVals$: $\langle column_{index}$, [list of $[x_i]values]\rangle$

1: Map the T_{RDD} to work on the p $T_{RDD_{(i)}}$ generated partitions
2: Index each column using the $zipWithIndex()$ and the $swap()$ functions
3: Reduce using the $groupByKey()$ function

Algorithm 2 Calculate the mean and the median

Input: $ColumnsIndxVals$: $\langle column_{index}$, [list of $[x_i]values]\rangle$

Output: $MeanMedian$: $\langle column_{index}$, value$\rangle$

1: Map the $ColumnsIndxVals$
2: if $column_{index} == 0$ then
3: Calculate the median
4: return $\langle column_{index}$, median$\rangle$
5: else
6: Calculate the mean
7: return $\langle column_{index}$, mean$\rangle$

Once the $ColumnsIndxVals$ is ready, Sp-DCA calculates the mean and the median in a sequential way within the distributed $ColumnsIndxVals$ partition as presented in Algorithm 2.

The output of the algorithm is a list of keys representing each column index where the first column, $column_{index} = 0$, is used to calculate both of the PAMP and SS values for each antigen. The rest of the keys, $column_{index} \neq 0$, are used to calculate the DS values. At this stage, the signal base for all antigens can be generated based on Algorithm 3.

Algorithm 3 Generate the signal base

Inputs: T_{RDD}, $MeanMedian$: $\langle column_{index}$, value$\rangle$

Output: $SignalBase$: $\langle key_{antigen}$, [PAMP, SS, DS]$\rangle$

1: Map the T_{RDD} to work on the p $T_{RDD_{(i)}}$ generated partitions
2: if $column_{index} == 0$
3: Calculate PAMP and SS values as explained in the itemized list and by using the $MeanMedian$ value
4: else
5: Calculate the DS value as explained in the itemized list and by using the $MeanMedian$ value

The output of this phase is a signal base where the first column represents the index of each antigen extracted from each $T_{RDD_{(i)}}$ and defined as a key, i.e. $key_{antigen}$. The list of signals [PAMP, SS, DS] of every $key_{antigen}$ is defined as a value. These $\langle$key, values$\rangle$ pairs define the $SignalBase$ output.

Context Assessment Phase. Based on the $SignalBase$ output and based on a DC population, the Sp-DCA processes its input signals to get three cumulative output signal values; namely the CSM, the $smDC$ and the mDC values as mentioned in Section 2.1. The calculations of these cumulative output signal values are achieved using Equation 3 as a signal processing equation, where $C = C_{[CSM,smDC,mDC]}$, and a set of weights:

$$C = \frac{((W_{PAMP} * PAMP) + (W_{SS} * SS) + (W_{DS} * DS))}{(W_{PAMP} + W_{SS} + W_{DS})} \tag{3}$$

$PAMP$, DS and SS are the input signal values obtained from the $SignalBase$ of category PAMP, danger and safe for all the antigens identified by their $key_{antigen}$. W_{PAMP}, W_{SS} and W_{DS} represent the weights used for PAMP, SS and DS, respectively. This equation is repeated three times, once per output signal. This is to calculate the interim output signal values for the CSM, the $smDC$ and the mDC outputs. These values are cumulatively summed over time [37].

These three DC output signals determine if an antigen type is anomalous or not. In fact, each DC in the population is assigned a migration threshold value (mt) upon its creation. So, if the value of CSM exceeds (mt) then the DC stops sampling signals; else the algorithm continues sampling and, also, keeps calculating and updating the values of CSM, $smDC$ and mDC. Once the cell has migrated, each DC forms a cell context that is used to perform anomaly detection in the classification of antigens. In fact, upon migration, the cumulative output signals are assessed and the greater of semi-mature or mature output signal becomes the cell context. This cell context is used to label antigens with the derived context value of 1 or 0. This process is presented in Algorithm 4.

Algorithm 4 Generate the contexts values

Inputs: *SignalBase*, *migrationThreshold*, *weights*

Output: *AntigenInitialContext*: $\langle key_{antigen}$, context$\rangle$

1: Map the *SignalBase*

2: Calculate the context of each antigen using the *migrationThreshold* and the *weights* as previously explained and by using Equation 3

The output of Algorithm 4, *AntigenInitialContext*, is the initial context of each antigen and it is in the form of ⟨key, value⟩ pairs. The key represents the index of each antigen, defined as $key_{antigen}$, and the value *context* is related to the initial context of each key and it takes a binary value; either 0 or 1. The initial context refers to the context of a single antigen as the latter is not copied m times yet. To further process this task, Sp-DCA has to generate the antigen

pool using the antigen multiplier m. Unlike the standard DCA which copies the antigens identifiers and construct a sequential pool from where random antigens are sampled, Sp-DCA proceeds as follows:

- First, Sp-DCA prepares an antigen pool in a distributed way based on the keys of the $SignalBase$; namely $key_{antigen}$ as presented in Algorithm 5. The output is a list of $key_{Antigen}$; each is multiplied m times.

Algorithm 5 Generate the antigen pool

Inputs: $SignalBase$: $\langle key_{antigen}, [PAMP, SS, DS]\rangle$, m: antigen multiplier

Output: $AntigenPool_{RDD}$: [List of $key_{Antigen}$]

1: Map the $SignalBase$
2: Generate m copies of each $SignalBase$ key from each $SignalBase$ mapped partition
3: Generate the $AntigenPool_{RDD_{(i)}}$ accordingly; where from every $SignalBase$ partition an $AntigenPool_{RDD_{(i)}}$ is constructed

- Second, Sp-DCA generates a context pool which is seen as m copies of every initial context generated in $AntigenInitialContext$ in Algorithm 4. This distributed job is presented in Algorithm 6.

Algorithm 6 Generate the context pool

Inputs: $AntigenInitialContext$: $\langle key_{antigen}$, context$\rangle$, m: antigen multiplier

Output: $ContextPool$: $\langle$index, [context, 0]$\rangle$

1: Map the $AntigenInitialContext$
2: Generate m copies of the value of each $AntigenInitialContext$ key
3: Create an index of each copy generated using the $zipWithIndex()$ and the $swap()$ functions
4: Map each index of each copy
5: Assign a value of 0 to every context

Technically and by mapping the *AntigenInitialContext*, Sp-DCA copies m times the value of each *AntigenContext* key, i.e. *context*. Then, the algorithm creates an index to these using both of the $zipWithIndex()$ and the $swap()$ functions in a way that the index defines the key of every multiplied context. After that and by mapping the generated indexes, Sp-DCA assigns to every initial context copy a value of 0. In this way, the value will be composed of a couple [context, 0]. The main idea behind this is to set a specific format for the contexts that once mapped with their corresponding random antigens, the context will be recognized via the 0 value and a cell context can be assigned to the sampled random antigens.

Classification Phase. Through the classification phase, Sp-DCA has to calculate the value for the cell context for all copies of the antigens to derive at the end the nature of the response by measuring the number of antigens that are fully mature. This process is presented in Algorithm 7.

Algorithm 7 Perform the classification task

Inputs: NumberIteration, *AntigenContext*, *ConttextPool*

Output: *Classification*: $\langle key_{Antigen}$, class$\rangle$

1: For each iteration i $\in$ [1, ..., NumberIteration] do
2: Generate *RandomAntigen* pool
3: Generate the random *AntigenContext* pool
4: Calculate $MCAVList$
5: End for
6: For j to $MCAVList$ size do
7: MCAV(antigen) = mean $MCAVList$(j)
8: End for
9: Map $MCAV$
10: Calculate the classification
11: Return $\langle key_{Antigen}$, class$\rangle$

To achieve this task, Sp-DCA has to go first through the derivation of the

set of the random antigens (line 2) and the calculation of their corresponding contexts (line 3). The generated context value is used to derive the Mature Context Antigen Value (MCAV). The MCAV is used to assess the degree of anomaly of a given antigen. The closer the MCAV is to 1, the greater the probability that the antigen is anomalous. The MCAV is calculated by dividing the number of times an antigen appears in the mature context by the total number of presentation of that antigen, i.e. m. Once the MCAV is calculated for each antigen, the algorithm can perform its classification task. However, as the Sp-DCA is based on a random selection of antigens then a loop is required to guarantee that in every iteration we will generate different antigens. As the random antigen sampling is not achieved in the context assessment phase, unlike the standard DCA which guarantees this random sampling with the use of multiple DCs, the loop becomes essential. Therefore, in every iteration the algorithm will generate an MCAV value for every antigen and generate the $MCAVList$ (line 4). As this process is repeated $NumberIteration$ times then a mean of the $MCAVList$ is calculated for each antigen in order to generate a single MCAV value (lines 6-8). Once this is achieved, Sp-DCA maps the obtained $MCAV$ collection and calculates the classification. This is done by comparing the MCAV of each antigen to an anomalous threshold (at). Those antigens whose $MCAVs$ are greater than (at) are classified into the anomalous category while the others are classified into the normal one.

The details on how to generate the $RandomAntigen$ pool, the $AntigenContext$ pool as well as the $MCAVList$ are as follows:

- **Generate the** $RandomAntigen$ **pool:** Sp-DCA has to generate first the list of the random antigens from where the random sampling will be performed. More precisely, through this task, Sp-DCA uses the $AntigenPool_{RDD}$ already generated in Algorithm 5 and applies the $Random.shuffle()$ operation in order to get a set of random antigens, i.e. a random list of $key_{Antigen}$, followed by the use of the $Parallelize()$ function to allow elements of the collection to be copied to form a distributed data set that can

be operated on in parallel. After that, an index of each random antigen is created using the $zipWithIndex()$ and the $swap()$ functions; resulting in ⟨index, $key_{Antigen}$⟩. On this new collection, a map is performed and a value equals to 1 is assigned to every $key_{Antigen}$. Hence, a ⟨key, value⟩ pair is constructed where the key is the index and the value is the couple [$key_{Antigen}$, 1]. The idea behind this is to set a default value of 1 to the antigen as if it appears by default in the mature context. Once the antigen is randomly mapped with its context, the value will be changed based on the value of the coupled context, i.e. if the context is 0 then the 1 value will change to 0 otherwise it is kept as 1. This distributed job is presented in Algorithm 8.

Algorithm 8 Generate the random antigens

Input: $AntigenPool_{RDD}$

Output: *RandomAntigen*: ⟨index, [$key_{Antigen}$, 1]⟩

1: $Random.shuffle$ the $AntigenPool_{RDD}$
2: $Parallelize$ the generated random collection
3: Create an index of each antigen using the $zipWithIndex()$ and the $swap()$ functions
4: Map each index of each antigen
5: Assign a value of 1 to every antigen

- **Generate the random** *AntigenContext* **pool:** To generate the context of every random antigen, Sp-DCA runs a distributed job which is presented in Algorithm 9. Sp-DCA applies first a union of both the *RandomAntigen*: ⟨index, [$key_{Antigen}$, 1]⟩ and the *ContextPool*: ⟨index, [context, 0]⟩ generated in Algorithm 8 and Algorithm 6, respectively. After that a $reducByKey()$ function is applied in a way to have an output in the form of ⟨index, [$key_{Antigen}$, context]⟩. More precisely, Sp-DCA will gather all elements having the same key, *index*, while replacing all the default 1 values previously assigned to the antigens with their correspond-

ing coupled context values. The recognition of the context is made via the 0 value which is assigned to it. After that, a map is applied to the latter collection and the *AntigenContext* is returned in the form of a ⟨key, value⟩ pair where the $key_{Antigen}$ is the key and the context is the value. At this stage, the m copies of the random antigens have different contexts assigned to them.

Algorithm 9 Generate the random antigens contexts

Input: *RandomAntigen*, *ContextPool*

Output: *AntigenContext*: ⟨$key_{Antigen}$, context⟩

1: Perform a *union* of both *RandomAntigen* and *ContextPool*
2: Reduce using the $reducByKey()$ function in a way to generate ⟨index, [$key_{Antigen}$, context]⟩ as explained in the itemized list (item 2)
3: Map the collection
4: Return ⟨$key_{Antigen}$, context⟩

- **Calculate the** $MCAVList$**:** This process is presented in Algorithm 10.

Algorithm 10 Generate the MCAV list

Input: *AntigenContext*

Output: $MCAVList$: ⟨$key_{Antigen}$, $List[MCAV]$⟩

1: Reduce the *AntigenContext* using the $groupByKey()$ function
2: Map the collection
3: Calculate $MCAV$ for each antigen
4: Fill the $MCAVList$

Technically, as the *AntigenContext* is generated where the m antigens have different contexts, a reduce $groupByKey()$ function is required to gather all the different contexts together for every key, i.e. $key_{Antigen}$. By applying this function, all the *AntigenContext* values, *context*, of the similar keys, $key_{Antigen}$, will be gather together and hence generating ⟨$key_{Antigen}$, $List[contexts]$⟩. After that a map is applied to the collection

and the MCAV is calculated by summing all the values in $List[contexts]$ which is filled with 0 and 1. The sum is then divided by the antigen multiplier m to finally generate the final MCAV for every antigen after being randomly multiplied, sampled and associated with the corresponding contexts. However, let us recall that this part is running $NumberIteration$ times within Algorithm 7 and hence generating $NumberIteration$ MCAVs for every antigen. At this stage, an $MCAVList$ is created where all the calculated MCAVs are stored.

3.3. Sp-DCA: A working example

We apply Sp-DCA to an example of a training data set presented in Table 1. The class "New Loan Decision" indicates if the client is allowed to have a new loan or not. In this context, the client is seen as an antigen.

Table 1: Training data set.

Client	**Age**	**Balance**	**Income**	**Previous loan amount**	**New Loan Decision**
Client-0	30	900	500	300	No
Client-1	36	2000	550	300	No
Client-2	22	300	350	150	No
Client-3	40	1200	700	800	Yes
Client-4	43	1800	800	900	Yes
Client-5	51	900	700	860	Yes

3.3.1. Initialization phase

Sp-DCA selects first of all some attributes and pre-categorizes them as PAMP, SS and DS. We suppose that we refer to the expert knowledge to map the features to their most appropriate signal types. From Table 1, the expert selects first the "Balance", the "Income" and the "Previous loan amount" features and categorizes them as PAMP, SS and DS. Specifically, the feature "Balance" is used to represent both of the PAMP and SS signals while the rest of the features are used to represent the DS.

3.3.2. Detection phase

In this phase, a signal database is generated. To achieve this task, Sp-DCA partitions the training data set into smaller splits where each will be handled in parallel. For this example, Sp-DCA works on two partitions, $p = 2$, where the first partition is composed of the first three antigens (client-0, client-1 and client-2) and the second one is composed of the rest of the instances. By applying Algorithm 1 at a first stage, the distributed steps work as follows:

- The two partitions are created using the map function:
  - For $p = 1$: by applying the $zipWithIndex()$ and the $swap()$ functions, the output is: [(0,900), (1,500), (2,300), (0,2000), (1,550), (2,300), (0,300), (1,350), (2,150)]
  - For $p = 2$: by applying the $zipWithIndex()$ and the $swap()$ functions, the output is: [(0,1200), (1,700), (2,800), (0,1800), (1,800), (2,900), (0,900), (1,700), (2,860)]
- By applying the $groupByKey()$ function, the output is the following: [(0,[900, 2000, 300, 1200, 1800, 900]), (2,[300, 300, 150, 800, 900, 860]), (1,[500, 550, 350, 700, 800, 700])]

We only present the detailed results related to each partition in Algorithm 1. For the rest of the algorithms the same reasoning is followed and we present results over the two p partitions. Once the base format is ready (the output of Algorithm 1), Algorithm 2 is applied to calculate the mean and the median:

- If $column_{index} == 0$ then the median is calculated generating: [0, 1050.0]
- If not then the mean is calculated generating: [1, 600.0]
- Same for $column_{index} == 2$ where the algorithm generates: [2, 551.66]

At this stage, the signal base is generated based on Algorithm 3 and as presented in Table 2.

Table 2: Signal data set.

Client (antigen)	PAMP	SS	DS
0	0	150	175.83
1	950	0	150.83
2	0	750	325.83
3	150	0	174.16
4	750	0	274.16
5	0	150	204.16

3.3.3. Context assessment phase

Through this phase, Sp-DCA has to generate first the contexts values using some specific parameters. It uses a threshold value $mt = 10$ for each used DC and a set of weights which is presented in Table 3.

Table 3: Example of weights used for signal processing.

	PAMP	SS	DS
CSM	2	2	1
smDC	0	0.7	0.7
mDC	2	-2	1

By applying Algorithm 4, we get the following contexts values where the key represents the antigen ID and the value represents the initial context of each antigen:

- [0, 0]
- [1, 1]
- [2, 0]
- [3, 1]
- [4, 1]
- [5, 0]

As each antigen is not copied multiple times yet, Sp-DCA has to generate the following antigen pool using the antigen multiplier $m = 3$ by applying Algorithm 5:

- [0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5]

It is important to recall that the generated output is distributed across the used machines and not in a single storage space. After that, Sp-DCA generates a context pool using Algorithm 6 via the following steps:

- The initial context is multiplied giving the following list: [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0]
- The $zipWithIndex()$ and the $swap()$ functions are applied generating the following set: [(0,0), (1,0), (2,0), (3,1), (4,1), (5,1), (6,0), (7,0), (8,0), (9,1), (10,1), (11,1), (12,1), (13,1), (14,1), (15,0), (16,0), (17,0)]
- By defining a specific format for the value, the output is as follows: [(0,(0,0)), (1,(0,0)), (2,(0,0)), (3,(1,0)), (4,(1,0)), (5,(1,0)), (6,(0,0)), (7,(0,0)), (8,(0,0)), (9,(1,0)), (10,(1,0)), (11,(1,0)), (12,(1,0)), (13,(1,0)), (14,(1,0)), (15,(0,0)), (16,(0,0)), (17,(0,0))]

3.3.4. Classification phase

Through the classification phase, Sp-DCA has to derive first the random antigens pool by applying Algorithm 8 via the following steps:

- By applying the $Random.shuffle()$ and the $parallelize()$ functions, the following distributed set is generated : [5, 1, 1, 2, 2, 3, 4, 3, 5, 4, 2, 0, 0, 0, 1, 3, 5, 4]
- The $zipWithIndex()$ and the $swap()$ functions are applied generating the following output: [(0,5), (1,1), (2,1), (3,2), (4,2), (5,3), (6,4), (7,3), (8,5), (9,4), (10,2), (11,0), (12,0), (13,0), (14,1), (15,3), (16,5), (17,4)]
- The output by defining a specific format for the value is as follows: [(0,(5,1)), (1,(1,1)), (2,(1,1)), (3,(2,1)), (4,(2,1)), (5,(3,1)), (6,(4,1)), (7,(3,1)), (8,(5,1)),

(9,(4,1)), (10,(2,1)), (11,(0,1)), (12,(0,1)), (13,(0,1)), (14,(1,1)), (15,(3,1)), (16,(5,1)), (17,(4,1))]

Then, the algorithm generates the random antigen context pool using Algorithm 9 via the following steps:

- By performing the union, the output is as follows: [(0,(5,1)), (1,(1,1)), (2,(1,1)), (3,(2,1)), (4,(2,1)), (5,(3,1)), (6,(4,1)), (7,(3,1)), (8,(5,1)), (9,(4,1)), (10,(2,1)), (11,(0,1)), (12,(0,1)), (13,(0,1)), (14,(1,1)), (15,(3,1)), (16,(5,1)), (17,(4,1)), [(0,(0,0)), (1,(0,0)), (2,(0,0)), (3,(1,0)), (4,(1,0)), (5,(1,0)), (6,(0,0)), (7,(0,0)), (8,(0,0)), (9,(1,0)), (10,(1,0)), (11,(1,0)), (12,(1,0)), (13,(1,0)), (14,(1,0)), (15,(0,0)), (16,(0,0)), (17,(0,0))]]

- By applying the $reduceByKey()$ function, the output is the following: [(0,(5,0)), (6,(4,0)), (12,(0,1)), (13,(0,1)), (1,(1,0)), (7,(3,0)), (14,(1,1)), (8,(5,0)), (2,(1,0)), (15,(3,0)), (3,(2,1)), (9,(4,1)), (4,(2,1)), (16,(5,0)), (10,(2,1)), (11,(0,1)), (17,(4,0)), (5,(3,1))]

At this stage, the 3 copies of the random antigens have different contexts assigned to them. Once this is achieved, the $MCAVList$ is generated based on Algorithn 10 as follows:

- By performing a $groupByKey()$ function, the output is as follows: [(0,[1, 1, 1]), (1,[0, 1, 0]), (2,[1, 1, 1]), (3,[0, 0, 1]), (4,[0, 1, 0]), (5,[0, 0, 0])]
- By calculating the MCAV, we get the following results:
  - [0, 1]
  - [1, 0.33]
  - [2, 1]
  - [3, 0.33]
  - [4, 0.33]
  - [5, 0]

Let us recall that all of these processes are repeated 10 times filling the MCAV List as highlighted in Algorithm 7. In this example, we have presented results related to a single iteration. To perform classification, Sp-DCA calculates, for each antigen, the mean of the 10 MCAVs and compares the resulting value to an anomaly threshold which is set to 0.5. Results are presented in Table 4.

Table 4: Classification results: MCAVs.

Antigen Type	**Mean MCAV**
Client-0	0.53
Client-1	0.4
Client-2	0.46
Client-3	0.5
Client-4	0.53
Client-5	0.56

In this example, client-0, client-3, client-4 and client-5 are classified as anomalous which means that they are not allowed to have a loan. This is because their corresponding mean MCAVs are greater than the defined anomaly threshold. On the other hand, client-1 and client-2 are classified as normal. When comparing the results to the actual decisions in Table 1, the generated accuracy is 83.33% as the mis-classification occurs when classifying client-0.

4. Experimental setup

4.1. Used benchmark

To demonstrate the effectiveness of our proposed approach we require a big classification data set with a large number of instances as the advantage of the data as well as the antigen vector parallelization schemas will become more pronounced for data sets with a large set of antigens. We, therefore, chose the Supersymmetry Particles (SUSY) data set from the UCI machine learning repository [43]. The data has been produced using Monte Carlo simulations

for classification purposes to distinguish between a signal process which produces supersymmetric particles and a background process which does not. The data includes 5 million data items referring to the simulated collision events described through 19 features. The first feature refers to the class label feature (1 for signal, 0 for background) followed by 8 features which are kinematic properties (low-level features) then 10 features which are functions of the first 8 features; these are high-level features derived by physicists to help discriminate between the two classes. The data set is nearly balanced with 46% positive examples. Input features were standardized over the entire data set with mean zero and standard deviation one, except for those features with values strictly greater than zero; these we scaled so that the mean value was one. A more detailed report on the SUSY data set can be found in [44]. Aiming to investigate the scalability of our Sp-DCA, we have created 4 synthetic different versions of the SUSY data set by generating 10, 20, 30 and 40 million of instances of the original data set. We will denote these versions as SUSY10M, SUSY20M, SUSY30M and SUSY40M. The databases are named according to the number of antigens contained, i.e. SUSY10M is a database containing 10 million data items and SUSY5M is a database containing 5 million data items. These synthetic databases are created by performing a traditional statistical analysis on the SUSY data set. Based on this, a multidimensional random process is defined that will generate the 4 bases with the same statistical characteristics as the SUSY data set. In such a way, we can guarantee that the multivariate relationship between the variables of the SUSY data set are preserved and hence the bases are fit to the original data enabling the creation of a realistic behavior to test the scalability of our Sp-DCA.

4.2. Testbed

Our experiments are performed on the High Performance Computing Wales[7] (HPC Wales) which provides a distributed parallel computing facility in support

[7] https://www.supercomputing.wales/

of research activity within the Welsh academic and industrial user community. Under this testbed, we used dual 12 core Intel Westmere Xeon X5650 2.67 GHz CPUs and 36GB of memory to test the performance of our Sp-DCA which is implemented in Scala 2.11 within the Apache Spark 2.1.1 framework. A a preliminary version of the algorithm is given in [45]. The source code implementation of our proposed algorithm will be made available online after acceptance for repeatability and future use. The main aim of running our experimentation under such testbed is to demonstrate the scalability of our proposed Sp-DCA distributed solution as it should be applied to data sets of a large number of antigens unlike its standard sequential version, i.e. the DCA. Indeed, the proposed solution should guarantee that it should provide satisfactory classification accuracy within a distributed environment.

4.3. Parameters description

Through the Sp-DCA steps, a specific parameter setting is adopted which is as follows: in the initialization phase, the class and the 10 high-level features are selected among the total number of features as these are functions of the first 8 low-level features as explained in Section 4.1. The first and the second high-level features are used to be mapped as a PAMP and SS, respectively, while the rest of the features are used all together to represent the DS. During the context assessment phase, a set of weights is used to derive the cumulative values as presented in Equation 3 in Section 3.2.2. The weights are 2, 0, 2, 2, 2, 2, 1, 0.9 and -0.9 for $W_{PAMP,CSM}$, $W_{PAMP,smDC}$, $W_{PAMP,mDC}$, $W_{SS,CSM}$, $W_{SS,smDC}$, $W_{SS,mDC}$, $W_{DS,CSM}$, $W_{DS,smDC}$ and $W_{DS,mDC}$, respectively. The migration threshold of an individual DC is set to 10 to ensure this DC to survive over multiple iterations. Indeed, each data item is mapped as an antigen, with the value of the antigen equals to the data ID of the item. An antigen multiplier $m = 9$ is used to derived the $AntigenPool_{RDD}$ resulting in 45, 90, 180, 270 and 360 million antigens for the SUSY5M, SUSY10M, SUSY20M, SUSY30M and SUSY40M data sets, respectively. To perform anomaly detection in the classification phase, a threshold is applied to the MCAVs. The threshold is

set to 0.35. So, if the MCAV is greater than the anomaly threshold then the antigen is classified as anomalous, else it is classified as normal. The resulting classified antigens are compared to the labels given in the original data sets. Because of the randomness of the antigen sampling process, the mean MCAVs are generated across 10 runs where the latter refers to the number of iterations. Based on these settings, we run the algorithm on 1, 2, 3, 4, 8, 12, and 16 nodes on HPC Wales. All of the Sp-DCA parameters, e.g. weights, migration threshold, antigen multiplier, etc., are set based on a tuning process that generated the best parameters values that best fit the algorithm.

4.4. Experimental plan, check points and hypotheses

4.4.1. Experimental plan

Our analysis first focuses on the scalability of the algorithm that allows it to solve the standard DCA inadequacy to be applied to big data. To do so, we will evaluate the performance of Sp-DCA on the 4 synthetic generated data sets, using the *speed-up*, the *size-up* and the *scale-up* criteria introduced in [46]. Second, to guarantee that Sp-DCA maintains its classification objectives an analysis of the classification accuracy is performed. This is done only on the real SUSY5M data set. At this stage, let us recall that in [37], it was highlighted that the DCA is sensitive to the input class data order, i.e. the performance of the algorithm is only observed when the algorithm is applied to ordered-classes training data sets, i.e. all data items labeled as "normal" are followed by all data items labeled as "abnormal", else the DCA classification performance will decrease notably. Taking this into consideration, in our experiments, we have used the SUSY5M first as a non-ordered database, dubbed $SUSY5M_{NO}$, and then as an ordered database, dubbed $SUSY5M_{O}$, where all class 0 data items are followed by all class 1 items. In $SUSY5M_{O}$, two RDDs are created where the first RDD, $SUSY5M_{O,0}$, refers to the normal class data items and the second RDD, $SUSY5M_{O,1}$, refers to the anomaly data items. Both of these RDDs are passed in turn to Sp-DCA where $SUSY5M_{O,0}$ is processed at first followed by the process of $SUSY5M_{O,1}$. This is to guarantee the respect of the

order and to cope with the mentioned data order restriction. We then compare the classification performance of Sp-DCA on these two data sets, $SUSY5M_{NO}$ and $SUSY5M_{O}$, to further analyze the behavior of the algorithm in such cases, i.e. if it keeps its sensitivity aspect while being a distributed version or not. More experiments are conducted on Sp-DCA to investigate its classification performance where the algorithm is compared to a set of well-known state-of-the-art classifiers. These are discussed in the following Section.

4.4.2. Algorithm under comparison and adopted statistical methodology

In this study, we compare the classification results of Sp-DCA to (i) DT: CART algorithm for decision tree with Gini coefficient [47], (ii) NB: Naive Bayes algorithm with kernel density estimator [48], (iii) K-NN: K-nearest neighbor algorithm [49], (iv) LR: multinomial logistic regression [50], and (v) RDF: the distributed Random Forest classifier implementation provided in the Apache Spark framework (org.apache.spark.mllib.tree.RandomForest) with the following parameters: maxDepth=6, numTrees=300, featureSubsetStrategy='all' and impurity='gini'. We utilize standard grid search for hyperparameter optimization. This is to automatically get the best parameters values that best fit the used algorithms.

As our experimental study involves some algorithms that are non-deterministic, i.e. that may provide different results over multiple repeated runs, the use of statistical testing is mandatory. We, therefore, consider 30 independent runs for the stochastic algorithms and we report accuracy in terms of AUC (Area under the ROC curve) and F-Score. To investigate the significance of any observed difference in classification accuracy we perform Wilcoxon signed rank tests. We analyze the statistical difference of results with a 95% confidence level ($\alpha = 0.05$). The Wilcoxon signed rank test is a non-parametric test used for paired samples. The test is based on the ranks of the absolute difference in the values of each pair. A p-value that is greater than or equal to the significance α (0.05 by default) leads to H0 which means that there is no difference between the results in the used compared data. Hence, we accept H0 and we reject H1.

However, a p-value that is strictly less than α means the opposite. Based on these algorithms, we will conduct a comparative study to further analyze the Sp-DCA classification performance.

4.4.3. Check points and hypotheses

From the descriptions provided in Section 4.4.1 and 4.4.2, we highlight that our research is not devote do optimize the accuracy obtained with our Sp-DCA method over a specific problem. We focus our experiments (i) on analyzing the scalability of the proposed solution enabling it to deal with big data classification problems, (ii) on investigating the insights tied to the parallelization of the DCA, i.e. if it can address the classification performance bottleneck of the standard DCA or not, and (iii) on the analysis of the behavior of the proposed parallel system, specifically in terms of classification performance. The set of the hypotheses on which our experiments will be based on is summarized in Table 5.

Table 5: List of Hypotheses.

Hypotheses	**Check Points**
H1	The scalability of Sp-DCA is noticed on all sizes of the used databases: SUSY10M, SUSY20M, SUSY30M and SUSY40M.
H2	Sp-DCA is sensitive to the input class data order: $Sp-DCA_O$ generates better classification results in comparison to $Sp-DCA_{NO}$.
H3	The classification performance bottleneck of the DCA is not expected to be addressed by executing via the distributed streaming library.
H4	$Sp-DCA_O$ generates better classification results than the state-of-the-art classifiers, i.e. DT, NB, K-NN, LR and RDF.

5. Results and analysis

In the following, we discuss our results. We will investigate each of the hypothesis presented in Table 5 and extract the related conclusions.

5.1. Analysis of the scalability

In this Section, we evaluate the performance of Sp-DCA with respect to its speed-up, size-up and scale-up.

5.1.1. Analysis of the speed-p

We first consider the speed-up of Sp-DCA: we keep the size of the data set constant and increase the number of nodes. The speed-up of a system with m nodes is defined as [46]:

$$\text{speed-up(m)} = \frac{\text{runtime on one node}}{\text{runtime on } m \text{ nodes}}$$

We plot the average speed-ups needed to run a single iteration within Algorithm 7 (over the 10 iterations executed) and their corresponding average times in Figures 3 and 4, respectively.

As discussed in [46], an ideal parallel algorithm has linear speed-up, which is, however, difficult to achieve in practice due to communication cost and the fact that the slowest slave dominates the total execution time, i.e. the skew problem.

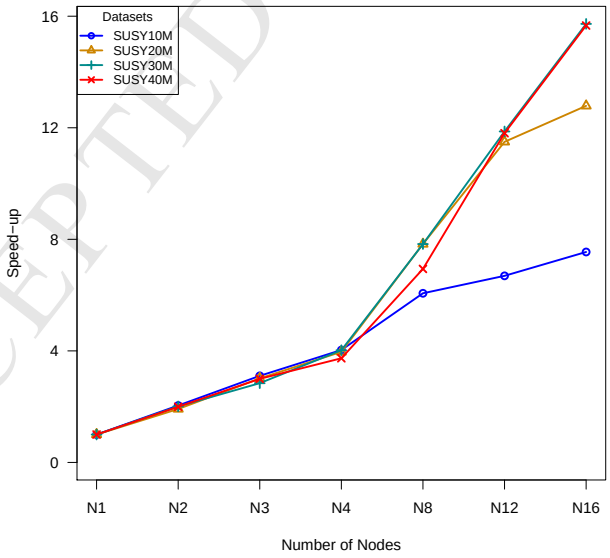


Figure 3: Speed-up results.

From Figure 3, we see that our method has a good speed-up performance. The more the size of the database increases, the more the speedup becomes closer

to linear. This performance is almost the same for databases with very different sizes, i.e. the SUSY20M, SUSY30M and SUSY40M. However, we notice that the SUSY10M database has a slightly lower speed-up curve. This is explained by the fact that based on the size of the SUSY10M data set the partitioning of the data is concentrated on the total number of nodes used resulting in a big communication cost. Therefore, the skew in this case is higher than in the other data sets and the total speedup is lower. Nevertheless, once the size increases starting from 20 million of instances we clearly observe a difference in the algorithm speed-up performance.

This observation is also supported by the execution times in Figure 4. The execution time quickly decreases with increasing the number of nodes while for the SUSY10M database we observe hardly any improvement when it comes to more than 4 nodes.

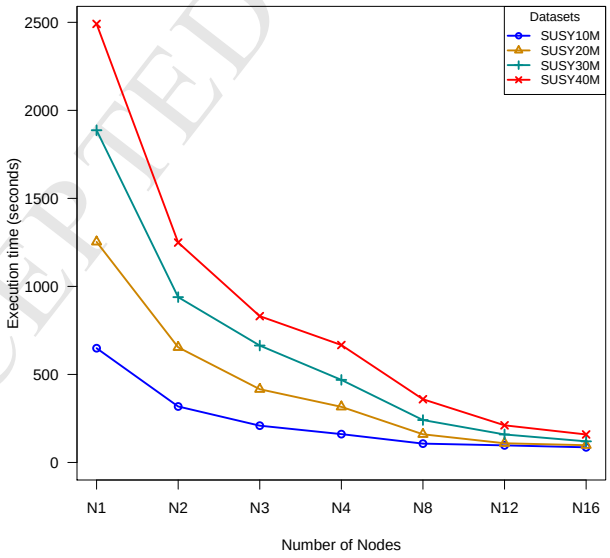


Figure 4: Average execution times.

5.1.2. Analysis of the size-up

The size-up measures how much the execution time increases as the data set is increased by a factor of m; defined as [46]:

$$\text{size-up(m)} = \frac{\text{runtime for data set of size } m \cdot s}{\text{runtime for baseline data set of size } s}$$

From Figure 5, we see that the size-up of Sp-DCA grows very quickly as the size of the base increases, but gets better as the number of nodes increases. Thus, the graph shows that our method has a very good size-up performance, i. e. that our method is able to process large data sets efficiently while keeping the number of nodes constant and increasing the size of the data. We can clearly see that a 2 times larger problem for instance needs about 2 times more time to run, e.g. SUSY40M needs 2 times more time to run than SUSY20M. This is noticed for all the used nodes.

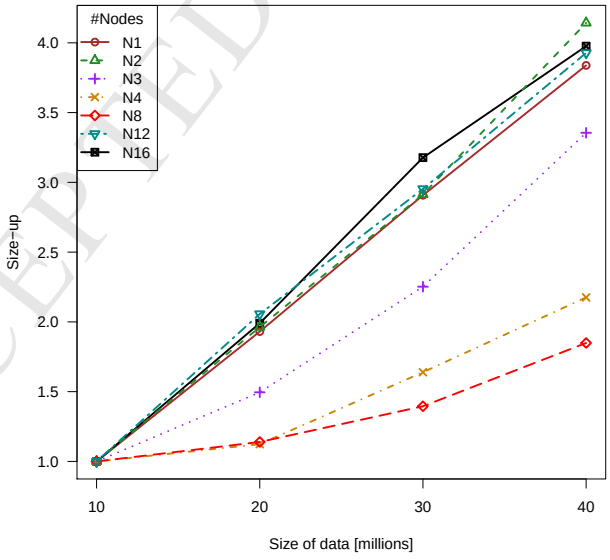


Figure 5: Size-up results.

5.1.3. Analysis of the scale-up

The scale-up measures the ability to grow both the system (number of nodes) and the database size. Scale-up is defined as the ability of an m-times larger system to perform an m-times larger job in the same run-time as the original system. The scale-up metric is defined as [46]:

$$\text{scale-up(m)} = \frac{\text{runtime for processing on 1 node}}{\text{runtime for processing } m \text{ data on } m \text{ nodes}}$$

To demonstrate how well our proposed Sp-DCA handles larger data sets when more nodes are available, we have performed scale-up experiments where we have increased the size of the databases in direct proportion to the number of nodes. For instance, for the data set SUSY10M, 10 million antigens are classified on 1 node and 40 million antigens (SUSY40M) are classified on 4 computers. Figure 6 shows the performance results of the databases. Clearly, the Sp-DCA scales very well as the scale-up values are all close to 1.

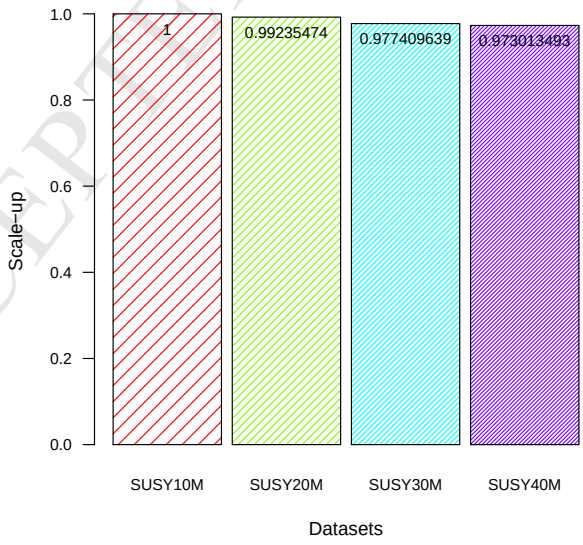


Figure 6: Scale-up results.

Based on the results obtained from the speed-up, the size-up and the scale-up, hypothesis H1 is accepted. This is presented in Table 6.

Table 6: Hypothesis 1 Result.

	Check Point	**Conclusion**	**Decision**
H1	The scalability of Sp-DCA is noticed on all sizes of the used databases: SUSY10M, SUSY20M, SUSY30M and SUSY40M.	The scalability of Sp-DCA is noticed on all sizes of the used databases and it becomes more significant when the size of the data set increases starting from 20 million of instances.	H1 is accepted.

5.2. Analysis of accuracy

To validate the suitability of our method with respect to classification, we investigate the classification performance of our Sp-DCA within a comparison to the set of classifiers presented in Section 4.4.2. We, also, investigate the behavior of our proposed parallel system with respect to the order of the classes as discussed in Section 4.4.1. To analyse this, we present results of 30 independent runs on the original data set with 5 millon instances in Table 7.

Table 7: Accuracy of the competing methods in terms of AUC and F-Score.

Algorithms	$Sp-DCA_O$	$Sp-DCA_{NO}$	DT	NB	K-NN	LR	RDF
AUC	72.92	50.48	69.00	69.90	66.40	72.10	76.55
F-Score	71.68	49.26	67.20	53.70	60.60	67.70	74.65

As previously mentioned, Sp-DCA is applied to the original non-ordered SUSY5M data set ($Sp-DCA_{NO}$), and to the ordered base ($Sp-DCA_O$). We name the algorithm $Sp-DCA_{NO}$ when it is applied in the former case, and $Sp-DCA_O$ when applied to the later case. This is to investigate if the Sp-DCA will maintain the standard DCA restriction to the input class data order, i.e. being sensitive to the input class data order and hence generating better classification results when applied to ordered data sets only (H2). Such analysis also permit to check if the classification bottleneck of the DCA is addressed or not via the execution within a distributed implementation design. We expect

that this limitation will not be solved (H3). From Table 7, we can clearly notice that the classification performance of $Sp-DCA_O$ in terms of both AUC (72.92%) and F-Score (71.68%) are much better than the results generated with $Sp-DCA_{NO}$ with 50.48% and 49.26% for AUC and F-Score, respectively. We, indeed, perform statistical tests as previously described in Section 4.4.2 and find that, based on the used evaluation metrics, the difference in the two algorithms is statistically signification at a confidence level of 0.05, i.e. p-value $(Sp-DCA_O,\ Sp-DCA_{NO}) < 2.2e^{-16}$. From these results, we can conclude that H2 is accepted and that the distributed framework has not solved the sensitivity problem which leads to the acceptance of H3.

On the other hand, comparing the classification performance of $Sp-DCA_O$ to the well-known state-of-the-art classifiers, and from Table 7, we notice that our proposed solution outperforms most of the competing classifiers, mainly DT, NB, K-NN and LR, in terms of classification performance. This is observed for both used metrics (AUC and F-Score). When, comparing $Sp-DCA_O$ to RDF, we notice that the later algorithm outperforms our algorithm in terms of AUC and F-Score. Statistical tests were also performed to further validate these conclusions where the resulting p-values were all less than $2.2e^{-16}$. Hence, the conclusions can be confirmed as the obtained results are statistically signification at a confidence level of 0.05. Based on this, H4 is partially accepted. A sum-up of the hypotheses is given in Table 8.

Table 8: Hypotheses 2, 3 and 4 results.

	Check Point	**Conclusion**	**Decision**
H2	Sp-DCA is sensitive to the input class data order: $Sp-DCA_O$ generates better classification results in comparaison to $Sp-DCA_{NO}$.	We conclude that Sp-DCA holds the sensitivity to the input class data order limitation as the classification accuracy of $Sp-DCA_O$ is much better than the one generated by $Sp-DCA_{NO}$.	H2 is accepted.

H3	The classification performance bottleneck of the DCA is not expected to be addressed by executing via the distrbuted streaming library.	As Sp-DCA kept its sensitivity to the input class data ordeer, as concluded in H2, the use of a distributed implementation design has not solved the bottleneck of the DCA as expected.	H3 is accepted.
H4	$Sp - DCA_O$ generates better classification results than the state-of-the-art classifiers; DT, NB, K-NN, LR and RDF.	We notice that $Sp - DCA_O$ outperforms almost all the used competing classifiers in terms of AUC and F-Score, except for RDF.	H4 is partially accepted.

6. Conclusion and future directions

In this paper, we have developed a distributed bio-inspired dendritic cell solution for large-scale data classification under the Spark framework, denominated as Sp-DCA. The Spark paradigm has offered an efficient environment to parallelize the functioning of the dendritic cell algorithm allowing it to overcome its memory and runtime restrictions. Focusing on the scalability of the algorithm, the experimental study carried out has shown that Sp-DCA has achieved good speed-up, size-up, and scale-up performances. Specifically, the scalability of Sp-DCA is noticed on all sizes of the used databases (SUSY10M, SUSY20M, SUSY30M and SUSY40M) and particularly the scalability becomes more significant when the size of the data set increases starting from 20 million of instances. In terms of classification performance, the experimental study carried out has shown that Sp-DCA holds its sensitivity characteristic to the input class data order as the algorithm classification accuracy in the ordered base case ($Sp - DCA_O$) is much better than the one of the unordered base case ($Sp - DCA_{NO}$). Based on this evaluation, we could emphasis that the classification performance bottleneck of the DCA is not addressed by the use of a distributed implementation design; as expected. Finally, when comparing the Sp-DCA classification results with well-known state-of-the-art algorithms we noticed that our proposed solution outperforms almost all the used classi-

fiers except random forest. Based on all conducted experiments and extracted conclusions, we highlight that Sp-DCA is an efficient distributed and scalable bio-inspired classification technique.

Our study provides many ideas for future research directions with particular focus on handling the Sp-DCA sensitivity aspect to the input class data order and on proposing a new distributed automated initialization phase for the algorithm. Moreover, tests on other real-world applications will demonstrate the wider applicability of our method.

7. Acknowledgments

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 702527. Additional thanks go to the support of the Supercomputing Wales project, which is part-funded by the European Regional Development Fund (ERDF) via the Welsh Government.

References

[1] C. P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on big data, Information Sciences 275 (2014) 314–347.

[2] M. Minelli, M. Chambers, A. Dhiraj, Big data, big analytics: emerging business intelligence and analytic trends for today's businesses, John Wiley & Sons, 2012.

[3] G. Bello-Orgaz, J. J. Jung, D. Camacho, Social big data: Recent achievements and new challenges, Information Fusion 28 (2016) 45–59.

[4] J. Archenaa, E. M. Anita, A survey of big data analytics in healthcare and government, Procedia Computer Science 50 (2015) 408–413.

[5] A. Elser, Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services, Springer Science & Business Media, 2012.

[6] A. Freitas, E. Curry, Big data curation, in: New Horizons for a Data-Driven Economy, Springer, 2016, pp. 87–118.

[7] H. Tong, Big data classification. (2014).

[8] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.

[9] J. Dean, S. Ghemawat, Mapreduce: A flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77. doi:10.1145/1629175.1629198.

[10] J. Schneider, M. Vlachos, Scalable density-based clustering with quality guarantees using random projections, Data Mining and Knowledge Discovery (2017) 1–34.

[11] P. Schäfer, Scalable time series classification, Data Mining and Knowledge Discovery 30 (5) (2016) 1273–1298.

[12] N. Talukder, M. J. Zaki, A distributed approach for graph mining in massive networks, Data Mining and Knowledge Discovery 30 (5) (2016) 1024–1052.

[13] W. Fan, A. Bifet, Mining big data: current status, and forecast to the future, ACM sIGKDD Explorations Newsletter 14 (2) (2013) 1–5.

[14] B. De La Iglesia, Evolutionary computation for feature selection in classification problems, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 3 (6) (2013) 381–407.

[15] D. Peralta, S. del Río, S. Ramírez-Gallego, I. Triguero, J. M. Benitez, F. Herrera, Evolutionary feature selection for big data classification: A mapreduce approach, Mathematical Problems in Engineering 2015.

[16] I. Triguero, S. del Río, V. López, J. Bacardit, J. M. Benítez, F. Herrera, Rosefw-rf: the winner algorithm for the ecbdl'14 big data competition: an extremely imbalanced big data bioinformatics problem, Knowledge-Based Systems 87 (2015) 69–79.

[17] I. Triguero, D. Peralta, J. Bacardit, S. García, F. Herrera, Mrpr: a mapreduce solution for prototype reduction in big data classification, neurocomputing 150 (2015) 331–345.

[18] I. Triguero, D. Peralta, J. Bacardit, S. García, F. Herrera, A combined mapreduce-windowing two-level parallel scheme for evolutionary prototype generation, in: Evolutionary Computation (CEC), 2014 IEEE Congress on, IEEE, 2014, pp. 3036–3043.

[19] V. López, A. Fernández, S. García, V. Palade, F. Herrera, An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics, Information Sciences 250 (2013) 113–141.

[20] I. Triguero, M. Galar, S. Vluymans, C. Cornelis, H. Bustince, F. Herrera, Y. Saeys, Evolutionary undersampling for imbalanced big data classification, in: Evolutionary Computation (CEC), 2015 IEEE Congress on, IEEE, 2015, pp. 715–722.

[21] I. Triguero, M. Galar, D. Merino, J. Maillo, H. Bustince, F. Herrera, Evolutionary undersampling for extremely imbalanced big data classification under apache spark, in: Evolutionary Computation (CEC), 2016 IEEE Congress on, IEEE, 2016, pp. 640–647.

[22] I. Triguero, M. Galar, H. Bustince, F. Herrera, A first attempt on global evolutionary undersampling for imbalanced big data, in: Evolutionary Computation (CEC), 2017 IEEE Congress on, IEEE, 2017, pp. 2054–2061.

[23] G. E. Batista, R. C. Prati, M. C. Monard, A study of the behavior of

several methods for balancing machine learning training data, ACM Sigkdd Explorations Newsletter 6 (1) (2004) 20–29.

[24] A. Verma, X. Llorà, D. E. Goldberg, R. H. Campbell, Scaling genetic algorithms using mapreduce, in: Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference On, IEEE, 2009, pp. 13–18.

[25] B. Wu, G. Wu, M. Yang, A mapreduce based ant colony optimization approach to combinatorial optimization problems, in: Natural Computation (ICNC), 2012 Eighth International Conference on, IEEE, 2012, pp. 728–732.

[26] A. W. McNabb, C. K. Monson, K. D. Seppi, Parallel pso using mapreduce, in: Evolutionary Computation, 2007. CEC 2007. IEEE Congress on, IEEE, 2007, pp. 7–14.

[27] P. Scully, Distribution of security information for industrial networks, Ph.D. thesis, Aberystwyth University (2016).

[28] E. Cantú-Paz, A survey of parallel genetic algorithms, Calculateurs paralleles, reseaux et systems repartis 10 (2) (1998) 141–171.

[29] E. Alba, J. M. Troya, et al., A survey of parallel distributed genetic algorithms, Complexity 4 (4) (1999) 31–52.

[30] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE transactions on evolutionary computation 6 (5) (2002) 443–462.

[31] E. Alba, Parallel evolutionary computations, Vol. 22, springer, 2006.

[32] E. Alba, Parallel metaheuristics: a new class of algorithms, Vol. 47, John Wiley & Sons, 2005.

[33] J. Greensmith, U. Aickelin, S. Cayzer, Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection, in: ICARIS, 2005, pp. 153–167.

[34] J. Greensmith, U. Aickelin, Articulation and clarification of the dendritic cell algorithm, in: ICARIS, 2006, pp. 404–417.

[35] J. G. Shanahan, L. Dai, Large scale distributed data science using apache spark, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2015, pp. 2323–2324.

[36] J. Greensmith, U. Aickelin, J. Twycross, Articulation and clarification of the dendritic cell algorithm, in: International Conference on Artificial Immune Systems, Springer, 2006, pp. 404–417.

[37] Z. Chelly, Z. Elouedi, A survey of the dendritic cell algorithm, Knowledge and Information Systems 48 (3) (2016) 505–535.

[38] J. Greensmith, M. B. Gale, The functional dendritic cell algorithm: a formal specification with haskell, in: Evolutionary Computation (CEC), 2017 IEEE Congress on, IEEE, 2017, pp. 1787–1794.

[39] S. Sakr, A. Liu, D. M. Batista, M. Alomari, A survey of large scale data management approaches in cloud environments, IEEE Communications Surveys & Tutorials 13 (3) (2011) 311–336.

[40] M. Snir, MPI–the Complete Reference: the MPI core, Vol. 1, MIT press, 1998.

[41] A. Fernández, S. del Río, V. López, A. Bawakid, M. J. del Jesus, J. M. Benítez, F. Herrera, Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 4 (5) (2014) 380–409.

[42] J. Dean, S. Ghemawat, Mapreduce: a flexible data processing tool, Communications of the ACM 53 (1) (2010) 72–77.

[43] A. Asuncion, D. Newman, Uci machine learning repository (2007).

[44] P. Baldi, P. Sadowski, D. Whiteson, Searching for exotic particles in high-energy physics with deep learning, Nature communications 5.

[45] Z. C. Dagdia, A distributed dendritic cell algorithm for big data, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018, 2018, pp. 103–104. doi:10.1145/3205651.3205701.
URL http://doi.acm.org/10.1145/3205651.3205701

[46] X. Xu, J. Jäger, H.-P. Kriegel, A fast parallel clustering algorithm for large spatial databases, in: High Performance Data Mining, Springer, 1999, pp. 263–290.

[47] J. R. Quinlan, Induction of decision trees, Machine learning 1 (1) (1986) 81–106.

[48] J. Su, H. Zhang, C. X. Ling, S. Matwin, Discriminative parameter learning for bayesian networks, in: Proceedings of the 25th international conference on Machine learning, ACM, 2008, pp. 1016–1023.

[49] N. S. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, The American Statistician 46 (3) (1992) 175–185.

[50] B. Krishnapuram, L. Carin, M. A. Figueiredo, A. J. Hartemink, Sparse multinomial logistic regression: Fast algorithms and generalization bounds, IEEE transactions on pattern analysis and machine intelligence 27 (6) (2005) 957–968.